CS422 - Programming Language Design

Operational Semantics

Grigore Roşu

Department of Computer Science
University of Illinois at Urbana-Champaign

Operational Semantics

By an *operational semantics* of a programming language, one typically understands a collection of rules specifying how its expressions, statements, programs, etc., are evaluated or executed. These rules say how a possible implementation of a programming language should "operate" and it is not difficult in practice to give an implementation of (an interpreter of) a language in any programming languages by just following and translating its operational semantics into the target implementation language.

There is no definite agreement on how an operational semantics of a language should be given, because any description of a programming language which is rigorous enough to quickly lead to a correct implementation of the language can be considered to be a valid operational semantics.

In this part of the course, we will investigate a particular but very common operational semantics approach, called *structural operational semantics* and abbreviated *SOS*; it is called "structural" because it is defined inductively over the structure of the syntax of the programming language under consideration.

There are two common, but conceptually rather different, SOS definitional styles of programming languages. They both consist of defining deduction (or inference, or derivation) systems for binary relations on *configurations*, where a configuration is typically a tuple containing some program fragment together with state infrastructure necessary to evaluate it.

- *Big-Step SOS*, also known as *natural semantics*. Under big-step SOS, the atomic "provable" entities are relations of configurations, typically written $C \Downarrow C'$, with the meaning that C' is the configuration obtained after the (complete) evaluation of C. A big-step SOS describes in a divide-and-conquer manner

how final evaluation results of language constructs can be obtained by combining the evaluation results of their syntactic counterparts (subexpressions, substatements, etc.). For example, the big-step SOS definition of addition in a language whose expression evaluation can have side effects is

$$\frac{\langle a_1, \sigma\rangle \Downarrow \langle i_1, \sigma_1\rangle,\ \langle a_2, \sigma_1\rangle \Downarrow \langle i_2, \sigma_2\rangle}{\langle a_1 + a_2, \sigma\rangle \Downarrow \langle i, \sigma_2\rangle} \text{ where } i \text{ is the sum of } i_1 \text{ and } i_2 \tag{1}$$

Here, the meaning of a relation $\langle a, \sigma\rangle \Downarrow \langle i, \sigma'\rangle$ is that arithmetic expression a is evaluated in state σ to integer i and new state σ'. If expression evaluation is side-effect-free, then one can drop the state from the right-side configuration and thus write big-step relations as $\langle a, \sigma\rangle \Downarrow \langle i\rangle$; big-step SOS

definitions transform as expected:

$$\frac{\langle a_1, \sigma\rangle \Downarrow \langle i_1\rangle,\ \langle a_2, \sigma\rangle \Downarrow \langle i_2\rangle}{\langle a_1 + a_2, \sigma\rangle \Downarrow \langle i\rangle} \text{ where } i \text{ is the sum of } i_1 \text{ and } i_2 \tag{2}$$

- *Small-Step SOS*, also known as *transitional semantics*, is how SOS was originally introduced. Under small-step SOS, the atomic provable entities are "one-computation-step" transitions, showing how a fragment of program is advanced *one step* in its evaluation process; a small-step SOS identifies for each language construct typically one of its syntactic counterparts which can be advanced precisely one step, and then shows how that sub-computation step translates into a one-computation step for the language construct. For example, the small-step SOS definition of addition is

$$\frac{\langle a_1, \sigma\rangle \to \langle a_1', \sigma\rangle}{\langle a_1 + a_2, \sigma\rangle \to \langle a_1' + a_2, \sigma\rangle} \tag{3}$$

$$\frac{\langle a_2, \sigma\rangle \to \langle a_2', \sigma\rangle}{\langle a_1 + a_2, \sigma\rangle \to \langle a_1 + a_2', \sigma\rangle} \tag{4}$$

$$\frac{\cdot}{\langle i_1 + i_2, \sigma\rangle \to \langle i, \sigma\rangle} \text{ where } i \text{ is the sum of } i_1 \text{ and } i_2 \tag{5}$$

We will be able to formally show that a step in a big-step SOS of a language corresponds to many steps in a small-step SOS of the same language.

It may be worth emphasizing that big-step and small-step SOS are two rather extreme types of SOS semantics. Many recent works on operational definitions of languages take the liberty to mix them; for example, one may evaluate the condition of a conditional statement in one big-step, but then transit to the left or to the right branch of the conditional in a small-step.

Syntax of a Simple Language

We will exemplify the two SOS definitional styles by means of a very simple non-procedural imperative language which has arithmetic and boolean expressions, conditionals and while loops.

A program is a sequence of statements followed by an expression. The expression is evaluated in the state obtained after evaluating all the statements and its result is returned as the result of the evaluation of the entire program. Arithmetic and boolean expressions do not have side effects; only statements can change the state.

Formally, the syntax of this simple language can be given as a context-free grammar (CFG) as follows:

$$
\begin{array}{rcl}
Var & ::= & \text{standard identifiers} \\
AExp & ::= & Var \mid 1 \mid 2 \mid 3 \mid ... \mid \\
 & & AExp + AExp \mid AExp - AExp \mid AExp * AExp \mid AExp / AExp \\
BExp & ::= & \mathsf{true} \mid \mathsf{false} \mid AExp \leq AExp \mid AExp \geq AExp \mid AExp == AExp \\
 & & BExp \ \mathsf{and} \ BExp \mid BExp \ \mathsf{or} \ BExp \mid \mathsf{not} \ BExp \\
Stmt & ::= & \mathsf{skip} \mid Var := AExp \mid Stmt \ ; \ Stmt \mid \{Stmt\} \\
 & & \mathsf{if} \ BExp \ \mathsf{then} \ Stmt \ \mathsf{else} \ Stmt \mid \mathsf{while} \ BExp \ Stmt \\
Pgm & ::= & Stmt \ ; \ AExp
\end{array}
$$

State, Configuration, Transition, Rule

An SOS defines a *transition relation* on *configurations*. In general, a configuration is a tuple containing a term over the syntax of the language and corresponding needed semantic infrastructure, such as a state, various control stacks, etc.; however, in our simple language definition we only need configurations consisting of pairs of a term and a state.

For this simple imperative language, a *state* is a map from variables to integer numbers $\texttt{Var} \to \texttt{Int}$. We let σ, σ', etc., denote states. If σ is a state and x a variable, then we let $\sigma[x]$ or $\sigma(x)$ denote the integer value to which σ maps x. Moreover, if x is a variable and i an integer, then we let $\sigma[x \leftarrow i]$ denote the function $\texttt{Var} \to \texttt{Int}$ defined as follows:

$$\sigma[x \leftarrow i](y) = \begin{cases} \sigma(y) & \text{if } x \neq y, \\ i & \text{if } x = y. \end{cases}$$

We also let $\emptyset$ denote the initial state. For simplicity, we here assume that all variables are initialized with 0 at the beginning of the computation. In other words, we consider that $\emptyset[x]$ is 0. Another possibility would be to consider states as *partial* functions and thus let $\emptyset[x]$ undefined, but this would create more cases to analyze in the subsequent SOS definitions.

In our simple language definition, we only need simple *configurations*. We enclose the various components forming a configuration with angle brackets. For example, $\langle a, \sigma \rangle$ is a configuration containing an arithmetic expression a and a state σ, and $\langle b, \sigma \rangle$ is a configuration containing a boolean expression b and a state σ. Configuration of different types need not necessarily have the same number of components. For example, since all programs

evaluate in the initial state, there is no need to mention a state next to a program in a configuration; in this case, a configuration is simply a one element tuple, $\langle p \rangle$, where p is a program. We will introduce configurations tacitly by need in our SOS language definitions; what distinguishes configurations of other structures are the angle brackets.

The core ingredient of an SOS definition is the *sequent*. Like in definitions of deductive systems in general, a sequent can be almost any tuple; however, in our particular SOS definitions, a *sequent* is typically a *transition*. We use arrows for transitions, such as $\Downarrow$ or $\rightarrow$. A transition takes a configuration to another configuration (the "next step"). For example, in a big-step SOS, a transition that signifies that arithmetic expression a evaluates in state σ to integer i can be written $\langle a, \sigma \rangle \Downarrow \langle i \rangle$ if the evaluation of expressions is side-effect free, or $\langle a, \sigma \rangle \Downarrow \langle i, \sigma' \rangle$ if the evaluation of expressions may have side effects, where σ' is the state after the evaluation of a

to i in σ. Also, in a small-step semantics, a transition $\langle s, \sigma \rangle \rightarrow \langle s', \sigma' \rangle$ signifies that statement s in state σ transits to statement s' (typically a variant of s whose computation is advanced by one step) in state σ'.

Finally, an SOS definition is a collection of *parametric rules* of the form:

$$\frac{\mathit{transition}_1,\ \mathit{transition}_2,\ \ldots, \mathit{transition}_k}{\mathit{transition}}$$

The intuition here is that *transition* is possible whenever $\mathit{transition}_1$, $\mathit{transition}_2$, ..., $\mathit{transition}_k$ are possible. We may also say that *transition* is *derivable*, or can be *inferred*, from $\mathit{transition}_1$, $\mathit{transition}_2$, ..., $\mathit{transition}_k$. This reflects the fact that an SOS definition can also be viewed as a logic system, where one can deduce possible behaviors of programs.

If $k = 0$, then we simply write

$$\frac{\cdot}{\mathit{transition}}$$

A Big-Step SOS Definition

Big-Step SOS Rules for Arithmetic Expressions

We here show how to derive transitions $\langle a, \sigma \rangle \Downarrow \langle i \rangle$, stating that the arithmetic expression a evaluates/executes/transits to the integer i in state σ. Note that in the case of our simple language, the transition relation is going to be *deterministic*, in the sense that whenever $\langle a, \sigma \rangle \Downarrow \langle i_1 \rangle$ and $\langle a, \sigma \rangle \Downarrow \langle i_2 \rangle$ can be deduced, then $i_1 = i_2$, because our language is deterministic. However, in the context of concurrent languages, $\langle a, \sigma \rangle \Downarrow \langle i \rangle$ states that a *may possibly* evaluate to i in state σ, but it may also evaluate to other integers.

We need to inductively define the transition relation for each language construct for arithmetic expressions. Since `Var` and `Int` are syntactic subcategories of `AExp`, we start by introducing the following two SOS rules, one for variables and the other for integers:

$$\frac{\cdot}{\langle x, \sigma \rangle \Downarrow \langle \sigma[x] \rangle} \tag{6}$$

$$\frac{\cdot}{\langle i, \sigma \rangle \Downarrow \langle i \rangle} \tag{7}$$

We next give the SOS rules for the arithmetic operations of addition, subtraction, multiplication and division. Recall that arithmetic and boolean expression do not have side effects, so the state σ does not change. However, the state needs to be carried as part of the configuration that appears to the left side of each sequent, because it may be needed if expressions contain variables, to look them up.

$$\frac{\langle a_1, \sigma \rangle \Downarrow \langle i_1 \rangle,\ \langle a_2, \sigma \rangle \Downarrow \langle i_2 \rangle}{\langle a_1 + a_2, \sigma \rangle \Downarrow \langle i \rangle} \text{ where } i \text{ is the sum of } i_1 \text{ and } i_2 \tag{8}$$

$$\frac{\langle a_1, \sigma\rangle \Downarrow \langle i_1\rangle,\ \langle a_2, \sigma\rangle \Downarrow \langle i_2\rangle}{\langle a_1 - a_2, \sigma\rangle \Downarrow \langle i\rangle} \text{ where } i \text{ is } i_1 \text{ minus } i_2 \qquad (9)$$

$$\frac{\langle a_1, \sigma\rangle \Downarrow \langle i_1\rangle,\ \langle a_2, \sigma\rangle \Downarrow \langle i_2\rangle}{\langle a_1 * a_2, \sigma\rangle \Downarrow \langle i\rangle} \text{ where } i \text{ is } i_1 \text{ times } i_2 \qquad (10)$$

$$\frac{\langle a_1, \sigma\rangle \Downarrow \langle i_1\rangle,\ \langle a_2, \sigma\rangle \Downarrow \langle i_2\rangle}{\langle a_1 / a_2, \sigma\rangle \Downarrow \langle i\rangle} \text{ where } i_2 \neq 0 \text{ and } i \text{ is the quotient of } i_1 \text{ by } i_2 \qquad (11)$$

Note that we chose not to "short-cut" the multiplication and the quotient operators when a_1 evaluates to 0. Therefore, in this case a_2 is still expected to produce a correct value in order for the rules to be applicable (e.g., a_2 cannot perform a division by 0).

Exercise 1 *Change the big-step SOS definition above so that multiplication and quotient short-cut when a_1 evaluates to 0.*

All rules above and below are *parametric*, that is, they can be

viewed as collections of concrete *instance rules*.

A possible instance of rule (8) can be the following, which, of course, seems problematic:

$$\frac{\langle 1, \sigma\rangle \Downarrow \langle 1\rangle,\ \langle 2, \sigma\rangle \Downarrow \langle 9\rangle}{\langle 1 + 2, \sigma\rangle \Downarrow \langle 10\rangle}$$

The rule above is indeed a correct instance of (8). However, one will never be able to infer $\langle 2, \sigma\rangle \Downarrow \langle 9\rangle$, so this rule cannot be applied in a correct inference.

The following is a correct inference, where x and y are any variables and σ is any state with $\sigma[x] = \sigma[y] = 1$:

$$\dfrac{\dfrac{}{\langle x,\sigma\rangle \Downarrow \langle 1\rangle}, \dfrac{\dfrac{}{\langle y,\sigma\rangle \Downarrow \langle 1\rangle}, \dfrac{}{\langle x,\sigma\rangle \Downarrow \langle 1\rangle}}{\langle y * x,\sigma\rangle \Downarrow \langle 1\rangle}, \dfrac{}{\langle 2,\sigma\rangle \Downarrow \langle 2\rangle}}{\langle y * x + 2,\sigma\rangle \Downarrow \langle 3\rangle}}{\langle x - (y * x + 2),\sigma\rangle \Downarrow \langle -2\rangle}$$

The proof above can be regarded as an upside-down tree, with dots as leaves and instances of SOS rules as nodes. We call such "complete" (in the sense that their leaves are all dots and their nodes are correct instances of SOS rules) trees *proof trees*. This way, we have a way to mathematically *derive facts* about programs within their SOS semantics. We may call the root of a proof tree the *fact that was proved or derived*, and the tree *its proof or derivation*.

The conditions which make a rule applicable, such as, for example, "where i is the sum of i_1 and i_2" in rule (8), are called *side conditions*. Side conditions typically constrain variables over

recursively enumerable domains (e.g., i, i_1 and i_2 range over integer numbers in rule (8). Therefore, each SOS rule comprises a *recursively enumerable* collection of concrete instance rules.

SOS Rules for Boolean Expressions

We can now similarly add transitions of the form $\langle b, \sigma \rangle \Downarrow \langle t \rangle$, where b is a boolean expression and t is a truth value in the set $\{true, false\}$:

$$\frac{\cdot}{\langle true, \sigma \rangle \Downarrow \langle true \rangle} \tag{12}$$

$$\frac{\cdot}{\langle false, \sigma \rangle \Downarrow \langle false \rangle} \tag{13}$$

$$\frac{\langle a_1, \sigma \rangle \Downarrow \langle i_1 \rangle,\ \langle a_2, \sigma \rangle \Downarrow \langle i_2 \rangle}{\langle a_1 \leq a_2, \sigma \rangle \Downarrow \langle true \rangle} \text{ where } i_1 \text{ less than or equal to } i_2 \tag{14}$$

$$\frac{\langle a_1, \sigma \rangle \Downarrow \langle i_1 \rangle,\ \langle a_2, \sigma \rangle \Downarrow \langle i_2 \rangle}{\langle a_1 \leq a_2, \sigma \rangle \Downarrow \langle false \rangle} \text{ where } i_1 \text{ larger than } i_2 \tag{15}$$

$$\frac{\langle a_1, \sigma\rangle \Downarrow \langle i_1\rangle,\ \langle a_2, \sigma\rangle \Downarrow \langle i_2\rangle}{\langle a_1 \geq a_2, \sigma\rangle \Downarrow \langle \mathit{true}\rangle} \text{ for } i_1 \text{ larger than or equal to } i_2 \quad (16)$$

$$\frac{\langle a_1, \sigma\rangle \Downarrow \langle i_1\rangle,\ \langle a_2, \sigma\rangle \Downarrow \langle i_2\rangle}{\langle a_1 \geq a_2, \sigma\rangle \Downarrow \langle \mathit{false}\rangle} \text{ where } i_1 \text{ smaller than } i_2 \quad (17)$$

$$\frac{\langle a_1, \sigma\rangle \Downarrow \langle i\rangle,\ \langle a_2, \sigma\rangle \Downarrow \langle i\rangle}{\langle a_1 \texttt{ equals } a_2, \sigma\rangle \Downarrow \langle \mathit{true}\rangle} \quad (18)$$

$$\frac{\langle a_1, \sigma\rangle \Downarrow \langle i_1\rangle,\ \langle a_2, \sigma\rangle \Downarrow \langle i_2\rangle}{\langle a_1 \texttt{ equals } a_2, \sigma\rangle \Downarrow \langle \mathit{false}\rangle} \text{ where } i_1 \text{ different from } i_2 \quad (19)$$

$$\frac{\langle b_1, \sigma\rangle \Downarrow \langle \mathit{true}\rangle,\ \langle b_2, \sigma\rangle \Downarrow \langle \mathit{true}\rangle}{\langle b_1 \texttt{ and } b_2, \sigma\rangle \Downarrow \langle \mathit{true}\rangle} \quad (20)$$

$$\frac{\langle b_1, \sigma\rangle \Downarrow \langle t_1\rangle,\ \langle b_2, \sigma\rangle \Downarrow \langle t_2\rangle}{\langle b_1 \texttt{ and } b_2, \sigma\rangle \Downarrow \langle \mathit{false}\rangle} \text{ where } t_1 \text{ or } t_2 \text{ is } \mathit{false} \quad (21)$$

$$\frac{\langle b_1, \sigma\rangle \Downarrow \langle \mathit{false}\rangle \;\; \langle b_2, \sigma\rangle \Downarrow \langle \mathit{false}\rangle}{\langle b_1 \;\texttt{or}\; b_2, \sigma\rangle \Downarrow \langle \mathit{false}\rangle} \tag{22}$$

$$\frac{\langle b_1, \sigma\rangle \Downarrow \langle t_1\rangle, \;\; \langle b_2, \sigma\rangle \Downarrow \langle t_2\rangle}{\langle b_1 \;\texttt{or}\; b_2, \sigma\rangle \Downarrow \langle \mathit{true}\rangle} \text{ where } t_1 \text{ or } t_2 \text{ is } \mathit{true} \tag{23}$$

$$\frac{\langle b, \sigma\rangle \Downarrow \langle \mathit{false}\rangle}{\langle \texttt{not}\; b, \sigma\rangle \Downarrow \langle \mathit{true}\rangle} \tag{24}$$

$$\frac{\langle b, \sigma\rangle \Downarrow \langle \mathit{true}\rangle}{\langle \texttt{not}\; b, \sigma\rangle \Downarrow \langle \mathit{false}\rangle} \tag{25}$$

Note that we chose not to shortcut the boolean operators either.

SOS Rules for Statements

Statements in our simple imperative language change the state, so we need to introduce a new transition relation of the form $\langle s, \sigma \rangle \Downarrow \langle \sigma' \rangle$, where s is a statement and σ, σ' are states. The SOS rules for statements are then:

$$\frac{\cdot}{\langle \texttt{skip}, \sigma \rangle \Downarrow \langle \sigma \rangle} \tag{26}$$

$$\frac{\langle a, \sigma \rangle \Downarrow \langle i \rangle}{\langle x := a, \sigma \rangle \Downarrow \langle \sigma[x \leftarrow i] \rangle} \tag{27}$$

$$\frac{\langle s_1, \sigma \rangle \Downarrow \langle \sigma'' \rangle, \ \langle s_2, \sigma'' \rangle \Downarrow \langle \sigma' \rangle}{\langle s_1; s_2, \sigma \rangle \Downarrow \langle \sigma' \rangle} \tag{28}$$

$$\frac{\langle s, \sigma\rangle \Downarrow \langle \sigma'\rangle}{\langle \{s\}, \sigma\rangle \Downarrow \langle \sigma'\rangle} \tag{29}$$

$$\frac{\langle b, \sigma\rangle \Downarrow \langle \mathit{true}\rangle,\ \langle s_1, \sigma\rangle \Downarrow \langle \sigma'\rangle}{\langle \texttt{if}\ b\ \texttt{then}\ s_1\ \texttt{else}\ s_2, \sigma\rangle \Downarrow \langle \sigma'\rangle} \tag{30}$$

$$\frac{\langle b, \sigma\rangle \Downarrow \langle \mathit{false}\rangle,\ \langle s_2, \sigma\rangle \Downarrow \langle \sigma'\rangle}{\langle \texttt{if}\ b\ \texttt{then}\ s_1\ \texttt{else}\ s_2, \sigma\rangle \Downarrow \langle \sigma'\rangle} \tag{31}$$

$$\frac{\langle b, \sigma\rangle \Downarrow \langle \mathit{false}\rangle}{\langle \texttt{while}\ b\ s, \sigma\rangle \Downarrow \langle \sigma\rangle} \tag{32}$$

$$\frac{\langle b, \sigma\rangle \Downarrow \langle \mathit{true}\rangle,\ \langle s, \sigma\rangle \Downarrow \langle \sigma''\rangle,\ \langle \texttt{while}\ b\ s, \sigma''\rangle \Downarrow \langle \sigma'\rangle}{\langle \texttt{while}\ b\ s, \sigma\rangle \Downarrow \langle \sigma'\rangle} \tag{33}$$

SOS Rules for Programs

Programs are always executed in the initial state, so we can define their SOS rule as follows:

$$\frac{\langle s, \emptyset \rangle \Downarrow \langle \sigma \rangle,\ \langle a, \sigma \rangle \Downarrow \langle i \rangle}{\langle s; a \rangle \Downarrow \langle i \rangle} \tag{34}$$

On Enumerability and Termination

Since each SOS rule comprises a *recursively enumerable* collection of concrete instance rules, and since a language definition consists of a finite set of ("parametric") SOS rules, by enumerating all the concrete instances of these rules we get a recursively enumerable set of concrete instance rules.

Furthermore, since proof trees built with nodes in a recursively

enumerable set are themselves recursively enumerable, it follows that the set of proof trees is recursively enumerable. In other words, we can find an algorithm that lists all the proof trees, in particular one that enumerates all the derivable transitions $\langle p \rangle \Downarrow \langle i \rangle$ for all programs p that "evaluate" to i. Note, however, that we only informally know the meaning of "evaluate" in the previous sentence. Formally, we say *by definition* that

- Program p *evaluates to* i iff the transition $\langle p \rangle \Downarrow \langle i \rangle$ is derivable, that is, iff there is a proof tree whose root is $\langle p \rangle \Downarrow \langle i \rangle$.
- Program p *terminates* iff there exists some integer i such that $\langle p \rangle \Downarrow \langle i \rangle$ is derivable.

Exercise 2 *Prove that the transition relation defined above is* deterministic, *that is:*

- *If* $\langle a, \sigma \rangle \Downarrow \langle i \rangle$ *and* $\langle a, \sigma \rangle \Downarrow \langle i' \rangle$ *derivable then* $i = i'$*;*
- *If* $\langle b, \sigma \rangle \Downarrow \langle t \rangle$ *and* $\langle b, \sigma \rangle \Downarrow \langle t' \rangle$ *derivable then* $t = t'$*;*

- *If a program p terminates then there is a unique i such that $\langle p \rangle \Downarrow \langle i \rangle$ is derivable.*

The above holds because our simple programming language is deterministic; we will see later in the class that concurrent programming languages may manifest non-deterministic behaviors (for example, due to data-races).

By enumerating all proof trees, one can eventually find such an i for any terminating program p. This simple-minded algorithm may take a very long time and a huge amount of resources, but it is theoretically important to understand that it can be done.

Exercise 3 *Show that there is no algorithm which takes as input a program p and says whether it terminates or not.*

The above holds because our simple language, due to its while loops, is Turing-complete. Thus, if one was able to decide termination of programs in our language then one was able to also

decide termination of Turing machines, which would contradict one of the basic undecidable problems, namely the *halting problem*.

An interesting observation here is that non-termination of a program corresponds to *lack of proof*, and that the latter is not decidable in many interesting logics. Indeed, in *complete* logics, that is logics that admit a complete proof system, one can enumerate all the truths. However, in general there is not much one can do about non-truths, because the enumeration algorithm will loop forever when run on a non-truth. In decidable logics one can enumerate both truths and non-truths; clearly, decidable logics are not powerful enough for our task of defining programming languages, exactly because of the halting problem argument above.

A Small-Step SOS Definition

Small-Step SOS Rules for Arithmetic Expressions

Unlike in a big-step SOS definition where one defines many (all) computation steps in one transition, in a small-step SOS definition a transition encodes only one step of computation. To distinguish small-step transitions from big-step ones, we use a plain arrow $\rightarrow$ instead of $\Downarrow$. The next two rules happen to be almost the same as in the big-step SOS semantics; that's because variable lookup and integer evaluation are one-step operations both under big-step and under small-step semantics. However, to avoid treating special cases in the subsequent small-step definitions, we prefer to mention the state in the right-side configuration even if it does not change:

$$\frac{}{\langle x, \sigma \rangle \rightarrow \langle \sigma[x], \sigma \rangle} \tag{35}$$

$$\frac{\cdot}{\langle i, \sigma\rangle \rightarrow \langle i, \sigma\rangle} \tag{36}$$

Exercise 4 *Suppose that rules that one wants to drop the state from all the right-side configurations of all the rules that do not change the state, including the two above. Is that possible to do that for all the rules? Rewrite the small-step SOS definition of our simple language maintaining minimal configurations int he right-sides of transitions. What is the drawback of this approach?*

We next give the small-step SOS rules for the arithmetic operations of addition, subtraction, multiplication and division:

$$\frac{\langle a_1, \sigma\rangle \rightarrow \langle a_1', \sigma\rangle}{\langle a_1 + a_2, \sigma\rangle \rightarrow \langle a_1' + a_2, \sigma\rangle} \tag{37}$$

$$\frac{\langle a_2, \sigma\rangle \rightarrow \langle a_2', \sigma\rangle}{\langle a_1 + a_2, \sigma\rangle \rightarrow \langle a_1 + a_2', \sigma\rangle} \tag{38}$$

$$\frac{\cdot}{\langle i_1 + i_2, \sigma\rangle \to \langle i, \sigma\rangle} \text{ where } i \text{ is the sum of } i_1 \text{ and } i_2 \tag{39}$$

$$\frac{\langle a_1, \sigma\rangle \to \langle a_1', \sigma\rangle}{\langle a_1 - a_2, \sigma\rangle \to \langle a_1' - a_2, \sigma\rangle} \tag{40}$$

$$\frac{\langle a_2, \sigma\rangle \to \langle a_2', \sigma\rangle}{\langle a_1 - a_2, \sigma\rangle \to \langle a_1 - a_2', \sigma\rangle} \tag{41}$$

$$\frac{\cdot}{\langle i_1 - i_2, \sigma\rangle \to \langle i, \sigma\rangle} \text{ where } i \text{ is } i_1 \text{ minus } i_2 \tag{42}$$

$$\frac{\langle a_1, \sigma\rangle \to \langle a_1', \sigma\rangle}{\langle a_1 * a_2, \sigma\rangle \to \langle a_1' * a_2, \sigma\rangle} \tag{43}$$

$$\frac{\langle a_2, \sigma\rangle \to \langle a_2', \sigma\rangle}{\langle a_1 * a_2, \sigma\rangle \to \langle a_1 * a_2', \sigma\rangle} \tag{44}$$

$$\frac{\cdot}{\langle i_1 * i_2, \sigma\rangle \to \langle i, \sigma\rangle} \text{ where } i \text{ is the product of } i_1 \text{ and } i_2 \tag{45}$$

$$\frac{\langle a_1, \sigma\rangle \to \langle a_1', \sigma\rangle}{\langle a_1/a_2, \sigma\rangle \to \langle a_1'/a_2, \sigma\rangle} \tag{46}$$

$$\frac{\langle a_2, \sigma\rangle \to \langle a_2', \sigma\rangle}{\langle a_1/a_2, \sigma\rangle \to \langle a_1/a_2', \sigma\rangle} \tag{47}$$

$$\frac{\cdot}{\langle i_1/i_2, \sigma\rangle \to \langle i, \sigma\rangle} \text{ where } i_2 \neq 0 \text{ and } i \text{ is the quotient of } i_1 \text{ by } i_2 \tag{48}$$

As in the case of "big step" rules, the "small step" SOS rules are also parametric. The following is a correct inference, where x and y are any variables and σ is any state with $\sigma[x] = 1$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\cdot}{\langle x, \sigma\rangle \to \langle 1, \sigma\rangle}}{\langle y * x, \sigma\rangle \to \langle y * 1, \sigma\rangle}}{\langle y * x + 2, \sigma\rangle \to \langle y * 1 + 2, \sigma\rangle}}{\langle x - (y * x + 2), \sigma\rangle \to \langle x - (y * 1 + 2), \sigma\rangle}$$

The above can also be regarded as a proof, but this time of the fact that replacing the second occurrence of x by 1 is a correct one-step computation.

Small-Step SOS Rules for Boolean Expressions

We can now similarly add transitions of the form $\langle b, \sigma\rangle \to \langle b', \sigma\rangle$, where b is a boolean expression and σ is a state:

$$\dfrac{\cdot}{\langle \mathit{true}, \sigma\rangle \to \langle \mathit{true}\rangle} \tag{49}$$

$$\frac{\cdot}{\langle false, \sigma \rangle \rightarrow \langle false \rangle} \tag{50}$$

$$\frac{\langle a_1, \sigma \rangle \rightarrow \langle a_1', \sigma \rangle}{\langle a_1 \leq a_2, \sigma \rangle \rightarrow \langle a_1' \leq a_2, \sigma \rangle} \tag{51}$$

$$\frac{\langle a_2, \sigma \rangle \rightarrow \langle a_2', \sigma \rangle}{\langle a_1 \leq a_2, \sigma \rangle \rightarrow \langle a_1 \leq a_2', \sigma \rangle} \tag{52}$$

$$\frac{\cdot}{\langle i_1 \leq i_2, \sigma \rangle \rightarrow \langle true, \sigma \rangle} \text{ where } i_1 \text{ smaller than or equal to } i_2 \tag{53}$$

$$\frac{\cdot}{\langle i_1 \leq i_2, \sigma \rangle \rightarrow \langle false, \sigma \rangle} \text{ where } i_1 \text{ larger than } i_2 \tag{54}$$

$$\frac{\langle a_1, \sigma \rangle \rightarrow \langle a_1', \sigma \rangle}{\langle a_1 \geq a_2, \sigma \rangle \rightarrow \langle a_1' \geq a_2, \sigma \rangle} \tag{55}$$

$$\frac{\langle a_2, \sigma\rangle \rightarrow \langle a_2', \sigma\rangle}{\langle a_1 \geq a_2, \sigma\rangle \rightarrow \langle a_1 \geq a_2', \sigma\rangle} \tag{56}$$

$$\frac{\cdot}{\langle i_1 \geq i_2, \sigma\rangle \rightarrow \langle \mathit{true}, \sigma\rangle} \text{ where } i_1 \text{ larger than or equal to } i_2 \tag{57}$$

$$\frac{\cdot}{\langle i_1 \geq i_2, \sigma\rangle \rightarrow \langle \mathit{false}, \sigma\rangle} \text{ where } i_1 \text{ smaller than } i_2 \tag{58}$$

$$\frac{\langle a_1, \sigma\rangle \rightarrow \langle a_1', \sigma\rangle}{\langle a_1 \texttt{ equals } a_2, \sigma\rangle \rightarrow \langle a_1' \texttt{ equals } a_2, \sigma\rangle} \tag{59}$$

$$\frac{\langle a_2, \sigma\rangle \rightarrow \langle a_2', \sigma\rangle}{\langle a_1 \texttt{ equals } a_2, \sigma\rangle \rightarrow \langle a_1 \texttt{ equals } a_2', \sigma\rangle} \tag{60}$$

$$\frac{\cdot}{\langle i \texttt{ equals } i, \sigma\rangle \rightarrow \langle \mathit{true}, \sigma\rangle} \tag{61}$$

$$\frac{\cdot}{\langle i_1 \texttt{ equals } i_2, \sigma\rangle \to \langle \mathit{false}, \sigma\rangle} \text{ where } i_1 \neq i_2 \tag{62}$$

$$\frac{\langle b_1, \sigma\rangle \to \langle b_1', \sigma\rangle}{\langle b_1 \texttt{ and } b_2, \sigma\rangle \to \langle b_1' \texttt{ and } b_2, \sigma\rangle} \tag{63}$$

$$\frac{\langle b_2, \sigma\rangle \to \langle b_2', \sigma\rangle}{\langle b_1 \texttt{ and } b_2, \sigma\rangle \to \langle b_1 \texttt{ and } b_2', \sigma\rangle} \tag{64}$$

$$\frac{\cdot}{\langle t_1 \texttt{ and } t_2, \sigma\rangle \to \langle t, \sigma\rangle} \text{ where } t_1,\ t_2,\ t \text{ are truth values s.t. } t \text{ is ``}t_1 \text{ and } t_2\text{''} \tag{65}$$

Note that the rule above has 4 instances wrt t_1, t_2 and t, each further parametric in σ; however, the state σ cannot influence the applicability of these 4 instance rules.

$$\frac{\langle b_1, \sigma\rangle \to \langle b_1', \sigma\rangle}{\langle b_1 \texttt{ or } b_2, \sigma\rangle \to \langle b_1' \texttt{ or } b_2, \sigma\rangle} \tag{66}$$

$$\frac{\langle b_2, \sigma\rangle \to \langle b_2', \sigma\rangle}{\langle b_1 \texttt{ or } b_2, \sigma\rangle \to \langle b_1 \texttt{ or } b_2', \sigma\rangle} \tag{67}$$

$\dfrac{\cdot}{\langle t_1 \texttt{ or } t_2, \sigma\rangle \to \langle t, \sigma\rangle}$ where t_1, t_2, t are truth values s.t. t is "t_1 or t_2" (68)

$$\frac{\langle b, \sigma\rangle \to \langle b', \sigma\rangle}{\langle \texttt{not } b, \sigma\rangle \to \langle \texttt{not } b', \sigma\rangle} \tag{69}$$

$\dfrac{\cdot}{\langle \texttt{not } t, \sigma\rangle \to \langle t', \sigma\rangle}$ where t and t' are opposite truth values (70)

Small-Step SOS Rules for Statements

The small-step SOS rules for statements are:

$$\frac{\cdot}{\langle \texttt{skip}, \sigma \rangle \rightarrow \langle \sigma \rangle} \tag{71}$$

$$\frac{\langle a, \sigma \rangle \rightarrow \langle a', \sigma \rangle}{\langle x := a, \sigma \rangle \rightarrow \langle x := a', \sigma \rangle} \tag{72}$$

$$\frac{\cdot}{\langle x := i, \sigma \rangle \rightarrow \langle \sigma[x \leftarrow i] \rangle} \tag{73}$$

$$\frac{\langle s_1, \sigma \rangle \rightarrow \langle s_1', \sigma' \rangle}{\langle s_1; s_2, \sigma \rangle \rightarrow \langle s_1'; s_2, \sigma' \rangle} \tag{74}$$

$$\frac{\langle s_1, \sigma \rangle \rightarrow \langle \sigma' \rangle}{\langle s_1; s_2, \sigma \rangle \rightarrow \langle s_2, \sigma' \rangle} \tag{75}$$

$$\frac{\langle s, \sigma\rangle \to \langle s', \sigma'\rangle}{\langle \{s\}, \sigma\rangle \to \langle \{s'\}, \sigma'\rangle} \tag{76}$$

$$\frac{\langle s, \sigma\rangle \to \langle \sigma'\rangle}{\langle \{s\}, \sigma\rangle \to \langle \sigma'\rangle} \tag{77}$$

$$\frac{\langle b, \sigma\rangle \to \langle b', \sigma\rangle}{\langle \texttt{if } b \texttt{ then } s_1 \texttt{ else } s_2, \sigma\rangle \to \langle \texttt{if } b' \texttt{ then } s_1 \texttt{ else } s_2, \sigma\rangle} \tag{78}$$

$$\frac{\cdot}{\langle \texttt{if } \mathit{true} \texttt{ then } s_1 \texttt{ else } s_2, \sigma\rangle \to \langle s_1, \sigma\rangle} \tag{79}$$

$$\frac{\cdot}{\langle \texttt{if } \mathit{false} \texttt{ then } s_1 \texttt{ else } s_2, \sigma\rangle \to \langle s_2, \sigma\rangle} \tag{80}$$

$$\frac{\cdot}{\langle \texttt{while } b\ s, \sigma\rangle \to \langle \texttt{if } b \texttt{ then } (s;\ \texttt{while } b\ s) \texttt{ else skip}, \sigma\rangle} \tag{81}$$

Exercise 5 *Can we conclude that* $\sigma = \sigma'$ *from the fact that* $\langle s, \sigma \rangle \rightarrow \langle s', \sigma' \rangle$ *is derivable?*

Exercise 6 *We have seen that the transition relation defined by our previous big-step SOS semantics was deterministic. Show that the transition relation defined by the small-step SOS semantics above is* not *deterministic. How can we change the SOS definition above so that the defined transition relation becomes deterministic?*

We will see shortly that the non-deterministic nature of the small-step transition relation does not affect the overall determinism of our language.

Small-Step SOS Rules for Programs

Programs are always executed in the initial state, so we can define their SOS rule as follows:

$$\frac{\cdot}{\langle s; a\rangle \rightarrow \langle s; a, \emptyset\rangle} \tag{82}$$

$$\frac{\langle s, \sigma\rangle \rightarrow \langle s', \sigma'\rangle}{\langle s; a, \sigma\rangle \rightarrow \langle s'; a, \sigma'\rangle} \tag{83}$$

$$\frac{\langle s, \sigma\rangle \rightarrow \langle \sigma'\rangle}{\langle s; a, \sigma\rangle \rightarrow \langle a, \sigma'\rangle} \tag{84}$$

Relating Big-Step and Small-Step SOS Definitions

Intuitively, a big-step transition is a sequence of small-step transitions. To formally capture the notion of "sequence of transitions", we define the following *complete transitive closure* of the small-step transition, written $C \rightarrow^{+} \langle V \rangle$, where C is a configuration and V is a value:

$$\frac{C \rightarrow \langle V, ... \rangle}{C \rightarrow^{+} \langle V \rangle} \tag{85}$$

$$\frac{C \rightarrow C',\ C' \rightarrow^{+} \langle V \rangle}{C \rightarrow^{+} \langle V \rangle} \tag{86}$$

The dots in the first rule above stay for any remaining part of the configuration; note that if a configuration contains a value, that is, if it is of the form $\langle V, ... \rangle$, then it cannot be derived anymore using

the small-step SOS rules.

We can now formally state and prove the following theorem:

Theorem. *For any program* p *and any integer* i, $\langle p \rangle \Downarrow \langle i \rangle$ *derivable iff* $\langle p \rangle \rightarrow^{+} \langle i \rangle$ *derivable.*

Exercise 7 *Prove the theorem above.*

Exercise 8 *Elaborate on how non-termination is reflected in a small-step SOS semantics.*

Big-Step vs. Small-Step: Advantages/Disadvantages

The theorem above tells us that the two styles of semantics achieve eventually the same objective, at least for the simple language that we considered: they tell formally when a program p evaluates to an integer i. So when do we choose one versus the other when we define a programming language? To answer this question, we need to first understand the advantages and the disadvantages of the two definitional styles.

Advantages of Big-Step SOS.

- When it can be given to a language, it is *easier to understand* because it relates syntactic entities directly to their expected results

- *More abstract*, more mathematical/denotational; therefore, one can more easily define and *prove properties* about programs; of course, because of the equivalence of the two semantics, one can

also use the transitive closure of the small-step SOS transition relation, but this would be less natural than using the big-step semantics. For example, try to prove the following using both big-step and small-step semantics:

Exercise 9 *Prove that the following programs are equivalent:*

1. `while` b s *and* `if` b `then` $(s;$ `while` b $s)$ `else skip`*;*
2. `while` b $($`while` b $s)$ *and* `while` b s

- Particularly useful when defining *type systems* of programming languages; there, values are replaced by their types.

Disadvantages of Big-Step SOS.

- It is not *executable*. One always needs to provide a result value or state to each language construct and then use the SOS rules to "check" whether that result value or state is indeed correct. Even if one can in principle implement an interpreter by disseminating the

big-step SOS rules, they are in fact intended to give a formal description of what is possible in a language, not to be executable

- Because of the above, it *avoids or hides non-termination* of programs by avoiding or hiding entirely the means by which one can state that a program does not terminate. Non-termination appears as "lack of proof" in the SOS system, but so does a program that performs a division by zero (or, in more complex languages, one that produces any runtime error).

- It is very inconvenient, if not *impossible, to define non-deterministic or parallel languages* using a big-step semantics.

Advantages of Small-Step SOS.

- It is *executable*. Indeed, one can start with a program to evaluate and keep applying SOS rules whose left configuration *matches*; to apply a small-step rule, one typically needs to call recursively the "execution" procedure on smaller fragments (the ones above the lines). Nevertheless, one can log all the successful applications of rules obtained this way and thus get an execution of the original program. It is relatively straightforward to implement an interpreter for a language by just following, almost blindly, a small-step SOS definition of that language.

- Thanks to the above, non-termination of programs results in non-termination of searching for a proof; however, programs that perform division by zero (or runtime errors in general) can still be evaluated step-by-step until the actual error takes place; then one can be given a meaningful error message.

- It *supports definitions of non-deterministic and/or parallel*

languages. In fact, our small-step SOS definition of the language above was non-deterministic; it just "happened" that the final results of evaluating expressions or programs were deterministic (but one needs to prove it).

Disadvantages of Small-Step SOS.

- *Less suitable for proving properties* about programs; however, if one can also give a big-step semantics of a language and prove it equivalent to the small-step semantics, then one can perform proofs using the latter.

- It is somehow *too low level and explicit*, which results into relatively large small-step language definitions; in particular, one needs to explicitly give all the "congruence" rule for ordinary operators such as addition, multiplication, etc.

- It has a *rigid computation granularity*.

Disadvantages of both Big-Step and Small-Step SOS

(1) They are *not modular*, in the sense that the addition of new statements to a language may require one to change many other rules corresponding to definitions of unrelated statements.

Homework Exercise 1 *Add a halt statement with syntax* `halt`(a) *(a is an arithmetic expression) to our language and give it both a big-step and a small-step SOS definition. Also and more importantly, rewrite the SOS rules, both big-step and small-step, that need to be changed. Comment on why we had to change these.*

Moreover, if one adds functions to our language that can return (abruptly) values, then one needs to add a function stack to the configurations. This way, all configurations will change, so all the existing rules will have to change. And the list of "inconvenient" features that can be added to the language can continue:

exceptions, break/continue in loops, threads, etc.

(2) *Neither of them provides an appropriate semantical foundation for concurrent languages*. Big-step SOS cannot be used to define any meaningful concurrent language, while small-step SOS gives only an *interleaving semantics*. An interleaving semantics regards executions or behaviors of concurrent programs as linear lists of actions. However, on parallel architectures, executions of parallel programs are *not* interleaved; for example, statements that have only local effect in different threads can be safely and are executed concurrently.

(3) They are both operational and syntax-driven, so they tell us close to *nothing about models* of languages. Models, which in the SOS world are considered "something else", are related to both denotational semantics and realizations of languages, including implementation for them.